(30) Priority: 20.06.94 US 262404

(43) Date of publication of application:
27.12.95 Bulletin 95/52

(84) Designated Contracting States:
**DE FR GB IT**

(71) Applicant: **AT&T Corp.**
**32 Avenue of the Americas**
**New York, NY 10013-2412 (US)**

(72) Inventor: **Agrawal, Prathima**
**40 Colchester Road**
**New Providence,**
**New Jersey 07974 (US)**

Inventor: **Trotter, John Andrew**
**420 River Road Apt. F9**
**Chatham,**
**New Jersey 07928 (US)**
Inventor: **Goldberg, Aaron Jay**
**380 South Stanworth Drive**
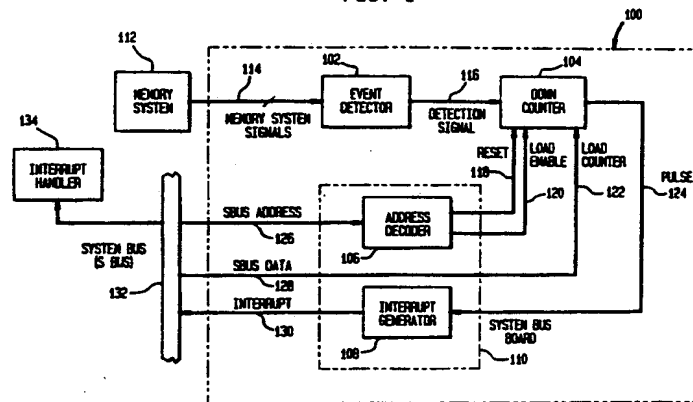**Princeton,**
**New Jersey 08540 (US)**

(74) Representative: **Watts, Christopher Malcolm**
**Kelway, Dr.**
**AT&T (UK) Ltd.**
**5, Mornington Road**
**Woodford Green**
**Essex, IG8 0TU (GB)**

(54) **Interrupt-based hardware support for profiling system performance**

(57) Fueled by higher clock rates and superscalar technologies, growth in processor speed continues to outpace improvement in memory system performance. Reflecting this trend, architects are developing increasingly complex memory hierarchies to mask the speed gap, compiler writers are adding locality enhancing transformations to better utilize complex memory hierarchies, and applications programmers are re-coding their algorithms to exploit memory systems. All of these groups need empirical data on memory behavior to guide their optimizations. This paper describes how to combine simple hardware support and sampling techniques to obtain such data without appreciably perturbing system performance. By augmenting a cache miss counter with a compare register and interrupt line such that the processor is interrupted when the counter marches the compare value, we can sample system state and develop cache miss profiles that associate cache misses with specific processes, procedures, call stacks, addresses, or user defined aspects of system state. This idea is implemented in the Mprof prototype that profiles data stall cycles, first level cache misses, and second level misses on the sun Sparc 10/41. Simple case studies are provided to illustrate Mprof's features.

FIG. 1

100

112

114

102 EVENT DETECTOR

116

104 DOWN COUNTER

MEMORY SYSTEM

MEMORY SYSTEM SIGNALS

DETECTION SIGNAL

134 INTERRUPT HANDLER

RESET
118

LOAD ENABLE
120

LOAD COUNTER
122

PULSE
124

SBUS ADDRESS
126

ADDRESS DECODER
106

SYSTEM BUS (S BUS)

SBUS DATA
128

132

INTERRUPT
130

INTERRUPT GENERATOR
108

SYSTEM BUS BOARD
110

Background of the Invention

### Field of th  Invention

5      The present invention relates generally to computer performance optimization and, more particularly, to performance profiling of computer systems.

### Related Art

10      Fueled by higher clock rates and superscalar technologies, growth in processor speed continues to outpace improvement in memory system performance. Reflecting this trend, various techniques have been implemented to compensate for this lag in performance. One technique, implemented by computer system architects, is the development of increasingly complex memory hierarchies to mask the lag in memory system performance. Another technique, implemented by compiler writers, is the addition of locality-

15   enhancing transformations to better utilize complex memory hierarchies. Still another technique, imple- mented by applications programmers, is the re-coding of algorithms to exploit memory systems.
      All of the above techniques require empirical data on memory system behavior to guide the particular optimization. Presently, there are a number of microprocessor and workstation vendors who provide hardware support for profiling memory system performance to meet this need for empirical data. For

20   example, the Intel Pentium microprocessor as well as the Sun ECache (external cache) controller in the Sun Sparc 10's and SparcCenter 1000's, feature cache miss count registers.
      While these simple counters provide a first step toward understanding memory system utilization, there are a number of drawbacks in utilizing cache miss count registers. In particular, simple cache miss counters do not directly relate cache behavior to the specific portion of the software which is incurring cache misses.

25   Instead, they only indicate the aggregate number of misses. This information, however, will not show which process, procedure, or data structure was responsible for the misses.
      What is needed, therefore, is a means for obtaining memory system performance data to develop cache miss profiles that associate cache misses with specific processes, procedures, procedure call stacks, and user defined aspects of the system state without interfering with system performance.

30

### Summary of the Invention

      The present invention is a system and method for profiling computer systems. The present invention is an implementation of a system profiler that samples system state using interrupts generated by a cache

35   miss counter supplemented with a compare register and interrupt line.
      One advantage of the present invention is that it has the ability to interrupt the processor when the cache miss count matches a predetermined value stored in the compare register. This improvement over conventional approaches enables the present invention to sample the state of the processor at a predetermined number of cache misses or cache stall cycles. The present invention is then able to develop

40   a cache miss profile that associates cache misses with specific processes, procedures, procedure call stacks, or even user defined aspects of system state.
      Another advantage of the present invention is its ability to provide significant assistance in isolating performance bottlenecks and guiding optimization of architectures, operating systems, compilers, and applications.

45   Still another advantage of the present invention is the implementation of a sampling approach that offers a better compromise between speed, accuracy, flexibility, portability, and extensibility than previous hardware and software methods for understanding memory system performance.
      Further features and advantages of the present invention, as well as the structure and operation of various embodiments of the present invention, are described in detail below with reference to the

50   accompanying drawings.

### Brief Description of the Figures

      In the drawings, like reference numbers indicate identical or functionally similarly elements. Additionally,

55   the left-most digit of a reference number identifies the drawing in which the reference number first appears. The present invention will be described with reference to the accompanying drawings, wherein:
      Figure 1 is a block diagram of the preferred embodiment of the interrupt-based hardware support system of the present invention;

Figure 2 is a block diagram of the alternative embodiment of the interrupt-based hardware support system of th present invention; and

Figure 3 is a per process cache profile for system-wide sampling of second level cache of a SUN Sparc 10/41.

## Detailed Description of the Preferred Embodiments

### A. Introduction

The present invention is a system and method for profiling computer system performance. The present invention may be implemented as a memory system profiler, referred to as Mprof, that samples system state using interrupts generated by a loadable decrementer cache miss counter supplemented with a compare register and interrupt line.

First, the use of memory system profiling and the reasons why conventional approaches for understanding memory system performance are inadequate are discussed. Then, the application of the present invention to profiling memory systems is given, with the preferred hardware implementation and alternative embodiments. Last, case studies are provided to illustrate how the profiles can be used to optimize applications for the cache hierarchy of a Sparc 10/41.

### B. Uses of Memory Profiles

Memory system profiles can provide significant assistance in isolating performance bottlenecks and guiding optimization of architectures, operating systems, compilers, and applications. Memory systems with multilevel cache hierarchies, write buffers, instruction prefetch units, banked memories, and sophisticated interconnection networks are becoming commonplace.

Commodity personal computers feature primary and secondary caches while cache-coherent multiprocessors are marketed as "desktop" workstations. The complex memory hierarchies in these systems are designed by architects who identify the memory reference characteristics of important benchmark applications and optimize the memory system to deliver performance for these key programs.

Once the architect has optimized the memory hierarchy for specific reference patterns, it is up to software developers to try to coerce applications into generating these patterns. Recent work has demonstrated that compiler transformations can increase data cache utilization by reordering and blocking loop nests, hide memory latency by carefully prefetching data before it is needed, and improve instruction cache performance by reordering basic blocks. For a discussion of compiler transformations which can increase data cache utilization by reordering and blocking loop nests see Wolf et al.," A Data Locality Optimizing Algorithm," in Proceedings of the ACM '91 Conference on Programming Language Design and Implementation (1991), pp. 30-44, herein incorporated by reference in its entirety. For a discussion of compiler transformations which can hide memory latency by carefully prefetching data before it is needed see Mowry, et al., "Design and Evaluation of a Compiler Algorithm for Prefetching," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (1992), pp. 62-73, herein incorporated by reference in its entirety. For a discussion of compiler transformations which can improve instruction cache performance by reordering basic blocks see McFarling, S., "Program Optimization for Instruction Caches," in Proceedings of the Third ASPLOS (1989), pp. 183-93; Pettis et al., "Profile Guided Code Positioning," in Proceedings of the ACM '90 Conference on Programming Language Design and Implementation (1990), pp. 16-27, herein incorporated by reference in its entirety.

Other work has shown that cache coherence traffic in shared memory multiprocessors can be reduced by supporting multiple coherence protocols and letting the compiler or operating system choose the most efficient protocol on a per page basis. See, for example, Veenstra et al., "A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (1992), pp. 149-60, herein incorporated by reference in its entirety. Moreover, many of the researchers developing software techniques for enhancing memory system utilization have explicitly noted that memory overhead profiles could be productively used to guide compilers in automatically applying their transformation. See, Mowry, et al., "Design and Evaluation of a Compiler Algorithm for Prefetching," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (1992), pp. 62-73; and Veenstra et al., "A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and

Operating Systems (1992), pp. 149-60 both of which are herein incorporated by reference in their entirety.

Finally, there is a large literature on rewriting applications to enhance locality and improve memory system performance. See, for example, Cheriton et al., "Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared Memory Multiprocessor: A First Experience," in Proceedings of the International Symposium on Shared Memory Multiprocessing, (1991) pp. 109-118, herein incorporated by reference in its entirety. See also, Rothberg et al., "Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations," in Proceedings of Supercomputing '90 (1990), pp. 232-41, herein incorporated by reference in its entirety. Given the existence of locality enhancing transformations, programmers need profiling tools to identify where a bottleneck is in the memory system and to quantify the extent of the bottleneck and to measure how transformations actually impact performance.

## C. Memory Profiling Techniques

The extensive set of applications for memory profiles described above has driven the development of several memory system profilers. These conventional memory profiles may be divided into three techniques or methods: (1) simulation-based techniques, (2) counter-based systems, and (3) software methods.

In simulation memory profiles, a memory reference trace of a program or set of programs is obtained, either by software instrumentation, microcode monitoring, or hardware reference tracing. An example of the software instrumentation approach may be found in Callahan et al., "Analyzing and Visualizing Performance of Memory Hierarchies," in Performance Instrumentation and Visualization, Simmons et al., Addison Wesley (1990), pp. 1-26, herein incorporated by reference in its entirety. An example of the microcode monitoring approach may be found in Agarwal et al., "New Technique for Capturing Address Traces Using Microde," in Proceedings of Thirteenth Annual International Symposium on Computer Architecture, (1986), pp. 119-27, herein incorporated by reference in its entirety. An example of the hardware reference tracing approach may be found in Lenoski et. al., "The DASH Prototype: Logic Overhead and Performance," in IEEE Transactions on Parallel and Distributed Systems (1993), vol. 4, no. 1, pp. 41-61, herein incorporated by reference in its entirety.

Once the memory reference trace of a program or set of programs is obtained in simulation memory profiles, the trace is then fed into an instrumented memory system simulator. The primary problem with simulation memory profiles is that it is slow, often increasing execution time by factors of 100-1000. The speed problem is exacerbated by the increasing complexity and size of modern memory systems. For example, the number of memory references needed to simulate the performance of a typical shared bus multiprocessors with 1-30 processors and 1-16 megabyte second level caches on each processor is large enough to make this approach impractical.

Because simulation is so slow, it is used primarily by architects to predict the performance of systems that cannot be measured because they have not yet been built. Yet, precisely because simulation is expensive, it is crucial that before simulating architects narrow the design space as much as possible by fully understanding the performance of today's systems. As mentioned above, to develop such an understanding, designers are including hardware for measuring cache miss rates and bus utilizations. The drawback of this counting hardware is that it does not give any direct insight into which interactions between the software and memory hierarchy lead to poor performance.

In order to establish a correspondence between hardware event counters and program events, software instrumentation must be added to access the counters. For example, the number of cache misses that occur can be measured in a procedure and on its behalf by reading the cache miss counter on entry and exit to the procedure and recording the difference in the counter value. However, this software technique has two important shortcomings. First, the instrumentation may perturb the system which is being measured. For a short procedure, the overhead of reading several hardware counters (e.g., read miss, write miss, bus utilization) on entry and exit and storing the totals may exceed the work actually done in the procedure. Second, if a context switch occurs between procedure entry and exit, the counter totals will include misses created by other processes.

Progress has been made in addressing some aspects of these shortcomings using model-based techniques to compensate for perturbation, adaptive instrumentation methods to reduce monitoring overhead, and operating systems support to create virtual per-process counters. For a discussion of the shortcomings in using model-based techniques to compensate for perturbation, see Malony et al., "Performance Measurement Intrusion and Perturbation Analysis," in IEEE Transactions on Parallel and Distributed Systems (1992), vol. 3, no. 4, pp. 433-50, herein incorporated by reference in its entirety. For a discussion of adaptive instrumentation methods to reduce monitoring overhead, see Reed et al., An Overview of the Pablo Performance Analysis Environment, Dept. of Computer Science, U. of Illinois, Unpublished Manu-

script, (1992), herein incorporated by reference in its entirety.

However, there r mains a fundamental intrusiveness problem whenever software instrumentation is inserted into frequently accessed regions of a program to gather fine-grained profiles using hardware counters. To get a sense of the magnitude of the resulting overhead of such a technique, the SPEC integer benchmarks are instrumented to access counters on procedure entry and exit. The overhead is measured in terms of impact on execution time. In practice, both a memory overhead profile and an execution time profile in a single run of the program, so perturbing execution time directly impacts the profile data. Also, it is expected that the profiling techniques will be applied to multithreaded programs where program behavior is highly dependent on the relative execution times of the threads. Therefore, monitoring techniques that impact execution time significantly will be suspect.

In order to measure overhead, the integer SPEC benchmarks are instrumented to record the dynamic number of procedure calls and the user mode execution time of each program. Given these statistics, and assuming the counter code at procedure entry and exit takes a fixed amount of time to execute, counter overhead can be estimated by:

$$2^*(\text{dynamic \# of procedure calls}) \cdot (\text{time for a counter execution})$$

It should be noted that this formula is conservative because it ignores several potential sources of overhead. For example, inserting counter code will expand the size of the executable, potentially degrading instruction cache performance. In addition, if the counters are inserted by the compiler, they will interact in an unpredictable way with the optimizer, affecting register allocation, instruction scheduling, and other optimizations.

Subject to these caveats, the above formula can be used to estimate overhead given a value for counter execution time. A generic counter update on procedure entry is represented by the pseudo-code:

Count__Array[Procedure__Index]- = Read__Stall__Cycle__Counter

The procedure exit code is similar, with the subtraction replaced by addition. The time to execute the pseudo-code can then be divided into counter update bookkeeping time and counter access time. On a RISC architecture, the update to the array involves at least a load, an add, and a store. In addition, to avoid multiple counter updates within a single recursive call, the update code must be protected with a counting semaphore. We refer to the cost of the counting semaphore and the counter update code as counter bookkeeping. On the Sparc 10/41 where the measurements were made, counter bookkeeping takes about 10 cycles at 40 Mhz or 0.25 microseconds. The cost of counter access is implementation dependent. If the counter is integrated on-chip with the microprocessor, it may cost only a cycle to access. If, on the other hand, the counter is implemented externally using a system bus (SBus) card, it takes around 30 cycles to read on the Sparc 10/41. Table 1 below illustrates the overhead results under these two assumptions about access time.

Table 1

| Instrumentation Overhead | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program Name | Dynamic Calls | Bookkeeping Overhead (sec.) | Access Overhead (sec.) | | Run Time (sec.) | % Overhead | |
| | | | 1 Cycle | 30 Cycle | | Fast | Slow |
| xlisp | 161206522 | 80.6 | 8.1 | 241.8 | 195.4 | 45 | 165 |
| gcc (cc1) | 58477688 | 29.2 | 2.9 | 87.7 | 129.6 | 25 | 90 |
| espresso | 17668163 | 8.8 | 0.9 | 26.5 | 57.1 | 17 | 62 |
| sc | 1749875 | 0.9 | 0.1 | 2.6 | 12.5 | 8 | 28 |
| eqntott | 4363768 | 2.2 | 0.2 | 6.5 | 32.4 | 7 | 27 |
| compress | 251472 | 0.1 | 0.0 | 0.4 | 2.4 | 6 | 21 |

Referring to Table 1, it can be seen that even with single cycle counter access, some programs experience 45% overhead; with slow counter access programs execute at 1.21 to 2.7 time their unin-

strumented speed. Also, it should be noted that the problem becomes more acute when we move from procedure level profiling to finer grained profiling of individual basic blocks or load instructions. In general, hardware counters do not seem to support non-intrusiveness, fine grained profiling.

5    There is an alternative approach to profiling, referred to as Mtool, that retains the speed of run-time monitoring without requiring hardware instrumentation on the memory system. Mtool isolates memory overhead by comparing the actual execution time of a program segment against the ideal execution time predicted assuming that all memory references are cache hits. For a further discussion of Mtool, see Goldberg et al., "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," in IEEE Transactions on Parallel and Distributed Systems (1993), vol. 4, no. 1, pp. 28-40,

10   herein incorporated by reference in its entirety. While this approach allows Mtool to quantify memory overhead on a per second basis, it does not provide information on what aspect of the memory hierarchy is being poorly utilized. For example, Mtool does not distinguish overhead due to primary instruction cache misses from overhead due to second level cache misses from write buffer stalls. From an optimiz r's perspective, distinguishing the individual sources of performance loss is critical to identifying appropriate

15   performance improving transformations.

Thus, previous memory system profilers have significant shortcomings that discourage their widespread use: simulation is too slow, hardware counters must be accessed via potentially intrusive software instrumentation, and software techniques like Mtool do not identify what part of the memory hierarchy is a bottleneck. These shortcomings have been alleviated with the sampling-based profiler of the pres nt

20   invention.

D. Sampling-Based Profilers

Sampling techniques are routinely used to build execution time profilers and are well known to those

25   skilled in the relevant art. Time profilers rely on an external timer to generate a clock interrupt. The interrupt is fielded by a handler that samples the system state, building up a statistical profile of where the syst m spends its time. For example, most UNIX implementations use the system timer interrupt to support a per process, program counter sampling service that is accessed via the profile system call and the prof tool. When the timer interrupt is serviced, if the interrupted process has requested profiling (with the profile call),

30   then a counter associated with the process' program counter is incremented. The program counter counts are then post-processed by prof to produce a table of per procedure execution times.

Assuming that the sampling is independent of the program execution, the sampled profile will provide a good estimate of the actual execution time spent in each procedure. A detailed discussion of the statistical significance of sampled profiles can be found in Ferrari, D., Computer Systems Performance Evaluation,

35   Prentice-Hall Inc., Englewood Cliffs, NJ (1978), Chapter 2, herein incorporated by reference in its entirety. A description of a robust UNIX implementation of code profiling is given in McCanne et al., "A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling," in Proceedings of 1993 Winter USENIX (1993), herein incorporated by reference in its entirety.

Moreover, because the sampling rate is relatively low (usually about 60 samples per second for

40   execution time profilers), sampling is less intrusive than inline software instrumentation. And finally, sampling techniques are not limited to execution time profiling. Hall et al., "Call Path Profiling of Monotonic Program resources in UNIX," in Proceedings of 1993 Summer USENIX (1993), describes a sampling-based profiler, Cpprof, which uses information provided by the UNIX System V /proc interface to profile with respect to a wide variety of system events including page faults, floating point exceptions, and bytes read

45   or written. Some previous hardware instrumentation has also supported profiling via sampling. In Cargill et al., "Cheap Hardware Support for Software Debugging and Profiling," in Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, (1987), pp. 82-3, for example, it is suggested that interrupts generated by a unit that counts instruction fetches to drive code profiling be used.

50   Another potential instance of interrupt-based profiling can be found in DEC's 21064 implementation of the Alpha architecture. This microprocessor includes two flexible performance counting registers that can be programmed to increment on a variety of events, generating an interrupt every $2^8$, $2^{12}$, or $2^{16}$ events. These events include misses in the on-chip instruction and data caches. However, the performance counters are apparently intended as a means of counting events over a long period of time relative to the event

55   frequency. See, for example, Digital Equipment Corporation, DECChip 21064-AA Microprocessor Preliminary Data Sheet. Maynard, MA (1992) pp. 3.20-3.22, herein incorporated by reference in its entirety. While the chip architects may have had simple aggregate counts in mind, the same interrupting mechanism can be used to build powerful sampling based profiling environments. Note that the 21064 mechanisms is not

7

ideal b cause it lacks fully programmable compare registers. This limitation prevents one from randomizing the sampling interval to insure statistical significance. Nevertheless, the profiling support in DEC's 21064 is vidence that vendors can implement sampling-based profiling in production systems.

Below, the requirements for the present invention are made explicit in the description of the preferred embodiment of the present invention. This embodiment counts primary and secondary cache misses and CPU data stall cycles. Alternative embodiments could easily be used to monitor other events like cache service time, TLB misses and write buffer stalls.

E. Hardware Sampling Implementation

1. Introduction

A basic problem in both debugging and performance profiling of systems such as those with multiple processors is monitoring the system without disturbing its operation. The present invention solves this problem by providing specialized hardware for monitoring certain operations of a processor and then interrupting the processor to record its state when the specialized hardware detects a predetermined condition. For example, a hardware timer interrupts the processor when waiting time at a particular address exceeds a preset threshold. In another example, multiple counters accumulate occurrences of waits in specified time ranges and interrupt the processor when one of the counts reaches a specified value.

Hardware monitors can track many aspects of memory system behavior. For example, by monitoring the external pins of a microprocessor that has an on-chip cache, it is usually straight-forward to detect misses in that cache. And, if the monitor is fabricated as part of the microprocessor (as in the Pentium and Alpha), misses in split on-chip can be identified as instruction or data misses. Regardless of the implementation of the event detector, the detection signal can be fed to a counter that triggers an interrupt when the count reaches a particular value. An interrupt handler can then record the state of the running system. If enough samples are recorded, this information will highlight the parts of a program or programs that are causing cache misses.

Referring to Figure 1, the performance monitor of the present invention is now described. The performance monitor 100 of the present invention consists of three main components: an event detection module 102, a loadable down-counter 104, and an interrupt generator 108. The event detection module 102 generates a pulse every time an event of interest occurs. The down-counter 104 counts the pulses generated by event detector 102 and causes interrupt circuitry 108 to generate an interrupt once the counter reaches zero. The initial counter value can be loaded so that an interrupt will be generated after the occurrence of a programmable number of events; this allows randomized sampling intervals to be used. In addition, by selecting an appropriate value for the loadable counter, relatively infrequent events will still interrupt the CPU a statistically significant number of times. The interrupt generator 108 implements the correct interface to generate CPU interrupts so the interrupt handler can record appropriate system state. In the preferred embodiment, performance monitor 100 utilizes program logic arrays (PALs) on an SBus card 110 to implement the event detector 102, down counter 104, and interrupt generator 108.

2. The Implementation of the Present invention

Referring to Figure 1, the present invention is implemented to monitor the behavior of memory system 112 of a Sun SPARC 10/41 workstation. The SPARC 10/41 offers an on-chip cache (split 10KB instruction and 16KB data) backed by an integrated 1MB second level cache which is connected to the main memory via a cache controller and a memory bus (MBus).

For primary cache misses, event detection circuit 102 detects a cache line read by monitoring several pins on the SRAM's of the second level cache that service the primary cache misses. This is represented by memory system signals 114 output by memory system 112. To detect second level cache misses, the event detector 102 monitors the memory bus (MBus) and outputs detection signal 116 on main memory reads received over memory system signal line 114. To properly monitor these main memory read signals, the present invention requires state information since MBus memory accesses use several bus cycl s. The event detector 102 may also detect data cache stall time by exploiting an external monitoring pin on the SuperSparc microprocessor that is active when the CPU pipeline is waiting on the data cache. This is also received via memory signal system line 114. In the preferred embodiment of the present invention, event detection circuit 102 is implemented in a PAL. However as one skilled in the relevant art would find apparent, detector 102 may be implemented in other types of hardware configurations.

Down counter 104 receives a detection signal generated by event detector 102 via detection signal line 116. Down counter 104 outputs a pulse on signal line 124 to interrupt generator 108 when a predetermined number of events has occurred. In the preferred embodiment of the present invention, down count r 104 is implemented using two PALs and is 20 bits deep. This enables performance monitor 100 to count up to $2^{20}$ events before a pulse is sent via pulse line 124 to interrupt generator 108. The interrupt generator 108 is implemented on an SBus card 110 and causes an SBus interrupt to occur on interrupt line 130 when it receives a pulse from down counter 104 via signal line 124. Thus, loadable down counter 104 and interrupt generator 108 of the present invention replace the simple cache miss counter used in conventional systems. Downcounter 104 includes a bit which indicates that the performance monitor system 100 generated the interrupt 130 and provides a register for clearing the interrupt. The SBus interface 132 is also used to reset and enable counter 104 via reset line 118 and load enable line 120, respectively. In addition, the SBus interface is used to load counter 104 with an predetermined value via an SBUS data line 128 which is passed through SBus board 110 and provided to the down counter 104 via load counter line 122. Down counter 104 generates a pulse on line 124 when the loaded number of detection signals received via detection signal line 116 is reached. In other words, the processor is interrupted when the cache miss count reaches zero.

Referring to Figure 2, an architectural block diagram of another preferred embodiment of the present invention is illustrated. Note that in the performance monitor 200 down counter 104 of Figure 1 has been replaced with a counter 204A and loadable compare register 205A to make the circuit more flexible.

This embodiment supports simultaneous profiling of primary and second level cache misses. This embodiment is an extension of the embodiment illustrated in Figure 1. The components of performance monitor 200 are implemented in field programmable gate arrays (FPGA) rather than PALs. By using an FPGA, it is possible to reconfigure the monitor circuit from software, allowing us to easily reprogram the monitor to trigger on different events or sets of events. The FPGA is integrated on an SBus board 210 with external probes, thereby providing a flexible monitoring architecture that can trigger on conditions involving state or detect accesses to specific address ranges (for example, I/O devices). The FPGA implements several separate counters 204A and 204B so that several distinct event classes can be monitored simultaneously, including primary and secondary level cache misses and data cache stall time. When any of the counters 204A,204B matches its compare value stored in the respective comparator 205A,205B, an interrupt is raised. That is, an interrupt is generated by the respective comparator 205A,205B over interrupt lines 224A,224B, respectively. The interrupt status bits indicate which of the monitored events caused the interrupt.

F. Recording System State

Upon receiving a profiling interrupt from the performance monitor of the present invention via SBus 132 or 232, the operating system and user application cooperate with the present invention to record profile data. Ideally, the present invention would sample enough system state to precisely identify the nature of every memory system bottleneck. In practice, the precision with which the sampled data describes the bottleneck must be balanced against the overhead of collecting large amounts of data. For example, while it may be desirable to know the state of a complex data structure when state is sampled, it is prohibitively expensive for a general purpose profiler to make a full dump of a process' data segment on each profiling interrupt.

The performance monitor of the present invention supports two data collection mechanisms that represent alternative trade-offs between flexibility, precision of information, and overhead. The most flexible and least expensive mechanism is system wide sampling. When system-wide profiling is enabled, each time a sampling interrupt occurs, the handler appends a brief sampled state record to a kernel profiling buffer. The record typically contains information such as:

(Timestamp, Current Process Id, Processor Status Word, Current Program Counter Value).

A user-level daemon periodically copies the profiling buffer to user space and saves its contents to a file. Post mortem tools can then synthesize cache miss profiles on a per-process basis, reporting user and system mode cache misses and plotting (using the timestamp data) cache miss samples versus time. In addition, if a mapping between Process IDs and corresponding executable files is maintained, the executables symbol tabl s may be used to synthesize UNIX prof style profiles that give cache miss counts on a per procedure basis.

The drawback of system wide sampling is that it provides only process and procedure-level information about the conditions under which cache misses are occurring. A preferred approach is the more flexible user level profiling implemented with software signals. In user level profiling, an application opens a special

sampling device to indicate to the interrupt handler located in the processor that it wishes to receive memory system sampling signals. When a sampling interrupt occurs, the interrupt handler checks wh ther the currently executing process has opened the sampling device, and if so, it posts a signal to th  process. When the kernel goes to resume the user process, it notices the posted signal and arranges to restart the

5     process in the signal handler that the user process has registered for the sampling signal. This handler records relevant aspects of the process state and then control returns to the process previously executing thread of execution. Note that this technique is not as non-intrusive as system-wide sampling because invoking the signal handler and resuming the previous thread of control involves extra overhead. However, as long as the sampling rate is not too high, the extra overhead will still be negligible.

10    The default interface to user level sampling in Mprof requires an application to be linked with a special library that replaces the default start and exit routines. The new start routine installs a sampling signal handler and opens the sampling device while exit writes the profile out to a file. (This library interface resembles linking with the -p flag to enable profiling under UNIX.) The Mprof sampling signal handler records the current procedure call stack so that cache misses can be associated with specific "contex-

15    tualized" procedure call paths. Mprof will, for example, report that foo2 incurs cache misses when called from foo1 called from main. Call path profiles are synthesized by feeding the call stack sample file produced by the handler into the Cpprof system which has facilities to highlight performance critical call paths and to interactively explore the data. Cpprof is discussed at length in Hall et al., "Call Path Profiling of Monotonic Program resources in UNIX," in Proceedings of 1993 Summer USENIX (1993), herein incor-

20    porated by reference in its entirety. An example is provided below which illustrates how user level Mprof profiles are viewed in Cpprof environment.

The inventors have found that Mprof's default call stack sampling signal handler 234 adequately isolates memory system bottlenecks. However, in more complex applications, the performance tuner may wish to replace the default handler with a specialized handler that records application specific information like

25    arguments to a particular procedure or the state of some global state variable. Because the signal handler is part of the application program, it is considered to be well known in the relevant art to replace it with a user defined handler.

Below are examples of profiles produced using Mprof's system wide and user level sampling.

30    G. Case Studies

The case studies given below are intended to illustrate the basic functionality of the present invention implemented to profile memory system performance. To emphasize the key features of Mprof, stylized synthetic benchmarks have been used when performing these case studies rather than full application

35    programs.

1. System Wide Sampling

This study examines the impact of multiprogramming on the effectiveness of the second level cache of

40    the Sparc 10/41. In particular, the behavior of a particular application, cache__hog, which repeatedly reads all the elements of a 0.9 MB data structure is profiled. A script that spawns a single instance of cache__hog, waits for it to complete, and then starts up two concurrent invocations of the program is shown below.

        cache__hog
        sleep 1
45      cache__hog &
        cache__hog;
        sleep 15

The actual data is collected by enabling the hardware sampling and record logging in the interrupt handler, running the test script, and then disabling sampling the commands are:

50      start__profd; gen__workload; stop__profd.

In this example, Mprof has been configured to generate a profiling interrupt after 16384 read misses in the second level cache. On each profiling interrupt in system wide sampling mode, the handler logs a timestamp, the process id (PID), and the program counter (PC). Mprof uses the timestamps and PIDs to plot cache miss samples versus time for each process. Figure 3 displays a plot for our test workload. In

55    Figure 4, the line 302 corresponds to the misses generated by the single invocation of cache__hog while lines 304 and 306 represent the misses of a pair of hogs sharing the machine. From Figure 3, it is clear that the miss rates of the multiprogrammed jobs substantially exceed that of their uniprogrammed counterpart. In particular, there are 10 samples taken when the program runs alone but when two executions share the

machine there are 75 + 77 = 152 samples. The additional misses are gen rated because the processes interfere in the cache, with each process flushing the other's data.

This phenomenon is not surprising, as each instance of *cache__hog* wants to use 90% of th lines in the second level cache. Mprof does, however, help us to visualize and quantify the effect. Moreover, the example illustrates how, over time, sampling correctly attributes misses to the processes that are incurring them. Note, however, that for multiprogrammed jobs, the samples seem to cluster into groups of four and five. This clustering is an artifact of poor sampling—for the purpose of illustration we have used a *fixed* sampling interval of 16384 misses. In any given 33 ms time slice, a process takes about 8000 second level cache misses. Thus, it is often the case that for a given time slice, we increment the miss counter by 8000 and then context switch so these misses are eventually charged to another process. In randomiz d sampling, such errors will tend to cancel, but in fixed interval sampling, correlations can arise producing the sort of clustering shown in Figure 3. In the pathological case, the sampling rate and miss count per-time slice can be chosen such that all samples will hit in one of two processes that are in fact missing equally often. This possibility argues strongly for randomizing the sampling by varying the inter-sample interval.

Returning to the profile results, the sampled data not only identifies which processes are missing in the cache, but also which program counters are associated with these misses. By periodically executing the processor status command (ps) during the profiling run, Mprof records a mapping from PIDs to executable file names. During the post mortem data viewing phase, when the user clicks on one of the lines of the summary graph given in Figure 3, Mprof exploits this record to map the PID corresponding to the line into an executable file name, extracts the symbol table from the file, and then synthesizes a procedure level cache miss profile from its sampled program counter information.

## 2. User Level Sampling in Cpprof

This case study illustrates Mprof's ability to relate data stall cycles to call paths, identifying regions of a program that map poorly onto the memory hierarchy. The case study compares several approaches to sorting a large array of records using the qsort library routine. This routine sorts an array of fixed size records given a pointer to the array, the size of a record, the number of records, and a function for comparing two records. On average, qsort performs NlogN compares when sorting N records; when two records are out of order they are swapped in place.

In this case study, the present invention operates on a 6 MB data set containing 200,000 32-byte records, each with a 4 byte integer key. As a base case, qsort is invoked on the full array of 32 byte records. This implementation is somewhat inefficient because qsort is comparing integer keys but swapping full 32 byte records. This unnecessary copying can be avoided by sorting pointers to records rather than the records themselves. Pseudo-code to initialize the pointer array and compare elements is shown below:

```
init_ptr_array(Record *rec, Record **record_ptr, int cnt) {
    for (i=0; i < cnt; i+=1)
        record_ptr[i] = rec + i;
}

compare_record_ptr(Record **rp1, Record **rp2) {
    return ( (*rp1)->key - (*rp2)->key );
}
```

The performance of the base case and record pointer sorts are compared by profiling both real time and data stall time via user-level sampling (real time signals are generating using the UNIX setitimer function). An excerpt from the Cpprof profile of the base case is shown below. Each line in the profile corresponds to a call path rooted at the qsort routine. Entries ending in %body% represent total time spent in a procedure body and entries ending in Data__Stall represent time spent stalled waiting for data. For example, the final call path in the profile below tells us that 0.62 seconds were spent waiting for data in the procedure compare__records when it was called from qsort.

format: Call Path [Time in Seconds]
-------------------------------------
(qsort %body%) [6.86]
(qsort Data_Stall) [1.98]


(qsort compare_records %body%) [1.40]
(qsort compare_records Data_Stall) [0.62]


Using the information in this profile, the first row of Table 2 is generated, which reports time the processor is active (CPU), time the processor is stalled waiting for data (DS) and total time. Note that the CPU figure is derived by subtracting DS from total time. Times are reported both for the qsort and compare routines individually and for qsort+compare. Looking at the qsort+compare columns, it is shown that eliminating the unnecessary copying by using the pointer sort saves 3.1 seconds of CPU time, but leaves data stall time unchanged. The likely explanation for this phenomenon is that the amount of data actually passing through the cache hierarchy is the same in both implementations. In the pointer based implementation, the comparison function dereferences a pointer to access the key field of a record. But since the primary cache has 32 byte lines, each reference to a key field causes the cache to load the entire 32 byte record.

The data bandwidth requirements can be reduced by building "little" records (LRecs) where each record contains only a pointer to a full database record and a copy of that record's key:

```
init_little_records(Record *rec, LRec *lrec, int cnt) {
    for (i=0; i < cnt;l i+=1) {
        lrec[i].key = rec[i].key;
        lrec[i].record_ptr = rec + i;
    }
}
```

These 8 byte "little" records can then be directly sorted using qsort. In this case, qsort will be swapping 8 byte records and comparing 4 byte keys. The extra 4 bytes of copying can be eliminated by sorting pointers to the little records rather than the records themselves, as was done above for full records. The results for an LRec and LRec pointer implementation appear in the bottom two rows of Table 2. Examining the time spent in qsort, it is shown that reducing the copying from 8 to 4 bytes saves nearly a factor of two in CPU time there (1.2 vs. 2.5 sec). However, it increases the CPU time in compare (there is an extra pointer dereference) and it increases data stall time in both qsort and compare. This increase in data stall time is explained by the fact that the array of pointers to records increases the active data set size from 8 bytes per little record to 12 bytes. In the end, sorting the little records themselves is the superior approach, running about 2.5 times faster than sorting full records.

Table 2

| Version | qsort time (sec) | | | compare time (sec) | | | qsort+compare (sec) | | |
|---|---|---|---|---|---|---|---|---|---|
| | CPU | DS | Total | CPU | DS | Total | CPU | DS | Total |
| Full Sort | 4.9 | 2.0 | 6.9 | 0.8 | 0.6 | 1.4 | 5.7 | 2.6 | 8.3 |
| Rec. Pointers | 1.1 | 1.0 | 2.1 | 1.5 | 1.6 | 3.1 | 2.6 | 2.6 | 5.2 |
| LRec | 2.5 | 0.1 | 2.6 | 0.6 | 0.1 | 0.7 | 0.2 | 0.2 | 3.3 |
| LRec Ptr | 1.2 | 0.7 | 1.9 | 1.5 | 1.0 | 2.5 | 2.7 | 1.7 | 4.4 |

The availability of data stall cycle times significantly simplified the above optimization process. In particular, the data access stall time information given above focused the attention of the inventors on reducing data movement in the full record pointer implementation. Also, in the LRec and LRec pointer implementations, there was a trade-off between CPU time and data stall time, and explicit measures of both

5  quantities allowed us to understand this trade-off. It is extremely difficult to understand such trade-offs without the sort of empirical data that Mprof provides. Note also that this example only touches on the combined functionality of Mprof/Cpprof. In the full system, the user currently has access to Mprof deriv d first and level cache miss information and data pipeline stall cycles as well as to Cpprof's real time, user time, and page fault profiles. In addition, Cpprof provides an interactive shell for synthesizing different types

10  of profile (function body, body + descendants, upward or downward call path) and relating program data to various subsets of the nodes in a program's dynamic call graph. In summary, the stack samples produced by Mprof fit naturally into an advanced profiling environment.

### H. Alternative Embodiments

15

The above case studies identify the type of information Mprof can extract. There are several features of Mprof which can be implemented to improve and extend its performance. For example, currently, the system state which is sampled is of a "code-centric" nature. That is, in both system-wide sampling and user level call path sampling, a program counter or set of program counters is being recorded. This data

20  allows one to identify where in the code the memory system performance is poor, but it does not directly record what data structures are at fault. By also recording the virtual address of the cache miss that triggers the sampling interrupt, one could get a crude profile of hot spots in the data. To create user level profiles from this data, however, the virtual addresses must be related back to user level data structures. Techniques for accomplishing such mappings are discussed in Martonosi et al., "MemSpy: Analyzing

25  Memory System Bottlenecks in Programs," in Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (1992), pp. 1-12, herein incorporated by reference in its entirety.

Sampling works well for long running programs where we can sample relatively infrequently and still accumulate enough samples to develop an accurate histogram of where performance is poor. However, in some circumstances a more fine-grained view may be required. For example, after obtaining a coarse

30  sampled profile of a multithreaded application, a programmer may want to focus on a specific procedure in a single thread. To better support such fine-grained profiling, we can add an enable bit to each Mprof counter such that the counter only increments when the enable bit is set. To determine the number of cache misses (or miss stall time if stall detectors are implemented) in a procedure, loop, or basic block, the program is as follows:

35  When program execution begins, *InitialValue* = read(counter0,*enable* = 0.

On entering the code segment, *enable* = 1.

On exiting the code segment, *enable* = 0.

On program exit, read the counter and subtract off *InitialValue*.

Counter overflow is handled by recording the number of overflow interrupts that the counter generates.

40  Note, this technique still has the problem that when a context switch occurs while executing an instrumented code segment, cache misses in other processes will be attributed to the instrumented code. As mentioned above, the operating system must maintain the state of the *enable* bit on a per process basis to avoid this problem.

45  ### I. Other Monitoring Applications

There are several natural extensions of the performance monitor 100 of the present invention as described above. One important extension is to instrument more of the memory hierarchy to provide better insight into where bottlenecks arise. In particular, this approach would distinguish between the sources of

50  memory overhead in today's processors, including translation look-aside buffers, on-chip instruction and data caches, and write buffers. Since this information lies within the microprocessor, hidden from external probes, chip vendor support is required for such profiling.

It should be noted that the sampling monitor technique of the present invention may also be applied to multiprocessor systems. This is particularly significant because parallel programs with many interacting

55  threads are easily perturbed by intrusive instrumentation techniques. Because sampling interrupts are relatively infrequent, the present invention is well-suited to the multiprocessor domain.

## 1. Network Monitoring

### a. Introduction

5      Another application of the present invention is in distributed systems where multiple processors connected by a communication network interact to solve a problem such as workstations connected by a high speed network. The most important feature which must be considered when optimizing system performance in this environment is understanding the interactions between the cooperating processors. The present invention is a network monitoring system that addresses this profiling problem in an inexpensive,

10     non-intrusive, and general purpose manner. This approach offers significant advantages over previous software instrumentation methods (that perturb program performance) and hardware monitors (that lack flexibility and generality).
       When used as a network monitor, the user to specify the conditions under which an interrupt is generated. By appropriately programming the monitor we can non-intrusively determine where and how a

15     system is waiting for data from the network, when network utilization is high or low, and where a program is generating significant network traffic.

### b) Detecting Synchronization Overhead

20     In any distributed system, one process may be waiting for another process to produce data. The problem is particularly serious because the effects of waiting may be cumulative. Processor A waits for data from B, which waits for data from C which may eventually slow down A. The conventional software approach to measuring synchronization overhead would require a clock before and after every network read to accumulate how long the program spends waiting for data to arrive. In fine-grained networks where

25     network packets can be as small as a single word, the overhead of these software clock accesses can dominate the actual data reading time, perturbing the system and eliminating the waiting times we are trying to monitor.
       The conventional hardware approach often used to circumvent this problem is to maintain a counter on the network board that accumulates read wait time. This approach allows us to determine the total time a

30     program spends waiting on network reads, but it does not tell us when during the program's execution it was waiting so it is difficult to isolate the problem.
       The approach of the present invention combines the read wait time counter with an interrupt generator. In the simplest case, the user can specify the value of a compare register and an interrupt is generated every time the read wait time counter reaches the compare value. The interrupt is then used by system

35     software to record aspects of the program's state. By default the interrupt handler can record the current procedure call stack, though we may customize the handler to record application specific state. This interrupt-based sampling technique has been used in the past to relate consumption of CPU time to program state, but it has not been applied more generally to resource classes like network read wait time or cache miss service time.

40

### c) Understanding Synchronization Overhead

       The present invention may also be used to gain information about the distribution of wait times on the network. For example, it would be useful to know how many long waits there were compared to the number

45     of shorter waits. To perform this function a configuration similar to that shown in Figure 2 can be used to accumulate individual wait times in specific ranges. The user programs the ranges and hardware interrupts whenever one of the ranger counters 204A,204B reaches its compare value in comparator 205A,205B, respectively. Note, even a single extent detector/counter pair is of substantial use in practice as the programmer often wants to distinguish whether synchronization overhead derives from a few long message

50     waits or many short waits.

### d) Resource Utilization

       The present invention can easily support traditional measurements like number of network transactions

55     and network utilization. But, due to the triggered interrupting capability, the present invention can relate these measurements to system state with little overhead. The present invention simply interrupts the program every time N units of resource are accumulated and record the program state. It is preferred that the present invention be configured to sample at random intervals with mean N units to avoid correlation

problems. Obtaining similar data by incrementing software counters on every network transaction would perturb the system, potentially changing the relative execution time of events on different processors, and affecting total system performance in an unpredictable way. Traditional hardware counters without an interrupting capability would also be inadequate because they do .not provide a mechanism for relating
5 resource usage to specific resource users.

## 2. Additional Applications

Finally, the monitor has applications beyond profiling and optimization.
10 Debug operations: The counters and decoder structures can be used to aid in debugging code that uses the communication system. For example, a count can be kept of the number of transmissions to and receives from each of the other processors in the system and this can be used to detect when and if the communication system becomes unsynchronized or deadlocked.

Sanity and error checking: The monitor can be used to detect error conditions. For example, if it is
15 known that no more than ten successive writes are done to a particular channel in a given time, it is possible to program the monitor to alert the processor if this rule is ever violated.

## 3. Network Interface Example

20 In this example, a network interface is considered which exists as a card in a workstation and allows the workstation to communicate using messages with the network. The waiting time, number of messages and the rate of communication can be monitored easily. Waiting time is monitored by counting the number of clock ticks that occur when the network is blocked and message rate is monitored by measuring the number of messages that are communicated in a given number of clock ticks. To illustrate how these
25 measures are useful, an analog circuit simulation is used as an example program. Circuit simulation has two main phases, the construction of a set of equations representing a circuit (model evaluation), and their solution The equations describing each element of the circuit (model equations) can be evaluated to determine the coefficients of the equations describing the entire circuit. Such evaluations can be performed independently and their results accumulated to derive the coefficients of the circuit equations. These
30 coefficients are stored in matrix form. The resulting matrix and a vector of variables define the set of equations representing the circuit. These equations can be solved using well known Lower Upper Decomposition (LUD). The LUD algorithm can be implemented on a parallel machine by first representing the LUD operations of row normalization, row update and element back-substitution, as tasks in a task graph. These tasks can then be scheduled on a machine with several processors. The results of a task may
35 be needed by tasks on other processors. The tasks and communications must be sequenced to insure that when a task is executed on a processor it has all the pertinent information the task needs to complete. The total LUD time is determined by the waiting time to access the network, time for communicating the data needed and the task execution time. A good schedule minimizes this total time.

The *waiting time* for the communication network is important during the LUD phase because the total
40 compute time is dependent on minimizing the total waiting time. A poorly sequenced task list will have a large waiting time because tasks that could otherwise execute have to wait for communications from other processors to communicate their results. Because our monitor counts events without perturbing the operation of the program it is possible to get an accurate value of the wait time. If the monitor is set to interrupt when the wait time exceeds a predetermined threshold the parts of the program that are causing
45 the most wait time can be identified easily.

The *number of communications* is important during model evaluation if the model equations are distributed to processors dynamically, depending on the computation load of a given processor. An unfavorable situation may arise if many communication intensive models are assigned to one processor. If conventional software monitoring techniques are used then the execution of the monitoring software may
50 change the assignment of model equations to processors and therefore would perturb the communications that occur.

The *rate of communication* is also important in model evaluation. In a multiprocessor environment model equations need to communicate their results with the processor that holds the matrix. Such communications can be interleaved with the model computation. If the amount of communication produc d
55 during device model evaluations exceeds the capacity of the network then the processors may stall waiting for the network to clear. By monitoring the rate of communications it is possible to identify the processors and parts of the programs sending data that cause these high rate.

4. Output queue examples

In this example we consider s veral workstations communicating with ach other using a n twork interfac card with a finite queue. When the queue is full communications cannot take place and the processor will stall. A monitor can be used to detect when the queue is full. The problem might be solved by interleaving other operations with the communications in this part of the program so that the communications occur less frequently.

a) Network blocking detection

It is important to know how much time an application is spending waiting to use a communication resource. This situation can occur when the communications of several processes running on a processor conflict with each other. Communication conflicts can be detected by monitoring the total wait time and comparing this with the wait time when the program is run using a network without any other load. A software monitor would perturb the execution time of the processes and therefore give an inaccurate time. The problem is exacerbated if the communications are frequent and small, then the monitoring software overhead has a greater effect as its execution time becomes a greater fraction of the total execution time of the original program. A hardware monitor will be able to monitor this blocking time without being intrusive, and only interrupt the processor when a significant wait time has elapsed, reducing the percentage of time spent in the monitoring software.

b) Platforms

We have described how the monitor would be used in an environment with many processors communicating using a communication network. The monitor can be applied in many other cases as well. For instance, it can be used in a uniprocessor to unobtrusively monitor many aspects of the system's performance such as disk access. It can also be applied in multiprocessor systems for any type of interconnection network. The idea is not just restricted to workstations it can be used in PCs and in mainframes alike.

**Claims**

1.  A system for profiling the behavior of memory systems in a processor, comprising:
    an event detector configured to generate a detection signal upon an occurrence of a predetermined event;
    a counter, coupled to said event detector, configured to receive and count occurrences of said detection signal received from said event detector, said counter further configured to generate a pulse signal when a predetermined count value of said detection signal occurrences is reached;
    an interrupt generator, coupled to said counter, configured to generate a profiling interrupt signal upon receipt of said pulse signal; and
    an interrupt handler residing in the processor, configured to record a state of the processor upon receipt of said profiling interrupt signal generated by said interrupt generator.

2.  The system of claim 1, wherein said event detector, said interrupt generator, and said counter are implemented in one or more PALs.

3.  The system of claim 1, wherein said counter is a down counter, said down counter generating said pulse signal when said count value is zero.

4.  The system of claim 1, wherein said counter is a loadable counter, said loadable counter configured to receive said predetermined count value.

5.  Th system of claim 1, wherein the memory system resides in a Sun SPARC 10/41 workstation.

6.  The system of claim 1, wherein said event decoder monitors a memory bus and outputs said detection signal when a main memory read signal is received to detect second level cache misses.

7. The system of claim 1, wherein said counter comprises a reset input configured to enable an external reset signal to reset said down counter.

8. The system of claim 1, wherein said counter comprises an enable input configured to enable an external enable signal to enable said down counter.

9. The system of claim 1, wherein said event detector detects data cache stall time, said counter counting a number of clock cycles during which said event detector receives a cache stall signal, said cache stall signal having a predetermined value when the processor pipeline is waiting on a data cache.

10. The system of claim 1, wherein said event decoder detects primary cache misses and second level cache misses.

11. The system of claim 1, wherein said counter further comprises an enable input, said counter configured to count said event when said enable bit is set.

12. A performance monitor system for simultaneous profiling of multiple conditions in a processor memory system, comprising:
a plurality of condition detectors, each of said plurality of condition detectors configured to monitor a predetermined condition and to generate a detection signal upon an occurrence of said predetermined condition;
a plurality of counters, each of said plurality of counters coupled to a respective one of said plurality of condition detectors, for receiving said detection signal and for generating a count signal indicating the number of said detection signals received;
a plurality of loadable compare registers, each of said plurality of loadable compare registers coupled to a respective one of said plurality of counters, configured to receive said count signal from said respective counter and for comparing said count signal to a predetermined value loaded in said each of said plurality of compare registers;
an interrupt generator, coupled to each of said plurality of comparators, configured to generate a profiling interrupt signal based upon a predetermined relationship between said pulse signals received from said plurality of comparators; and
an interrupt handler, residing in the processor, configured to record a state of the processor upon receipt of said profiling interrupt signal generated by said interrupt generator.

13. The system of claim 12, wherein the performance monitor is implemented in a field programmable gate array.

14. The system of claim 12, wherein each of said plurality of counters comprises an enable input, said enable input enabling each of said plurality of counters to count said event when said enable bit is set.

15. A network monitoring system for monitoring the interactions between the cooperating processors in a distributed system having multiple processors connected by a communication network, comprising:
an event detector configured to generate a detection signal upon an occurrence of a predetermined event;
a counter, coupled to said event detector, configured to receive and count occurrences of said detection signal received from said event detector, said counter further configured to generate a pulse signal when a predetermined count value of said detection signal occurrences is reached;
an interrupt generator, coupled to said counter, configured to generate a profiling interrupt signal upon receipt of said pulse signal; and
an interrupt handler residing in the processor, configured to record a state of the processor upon receipt of said profiling interrupt signal generated by said interrupt generator.

16. The system of claim 15, wherein said multiple processors are workstations.

17. The system of claim 15, wherein the performance monitor monitors the distribution of wait times on the network.

18. Th  system of claim 15, wherein said predetermined value is a random number, thereby causing the system to sample the network at random intervals.

5
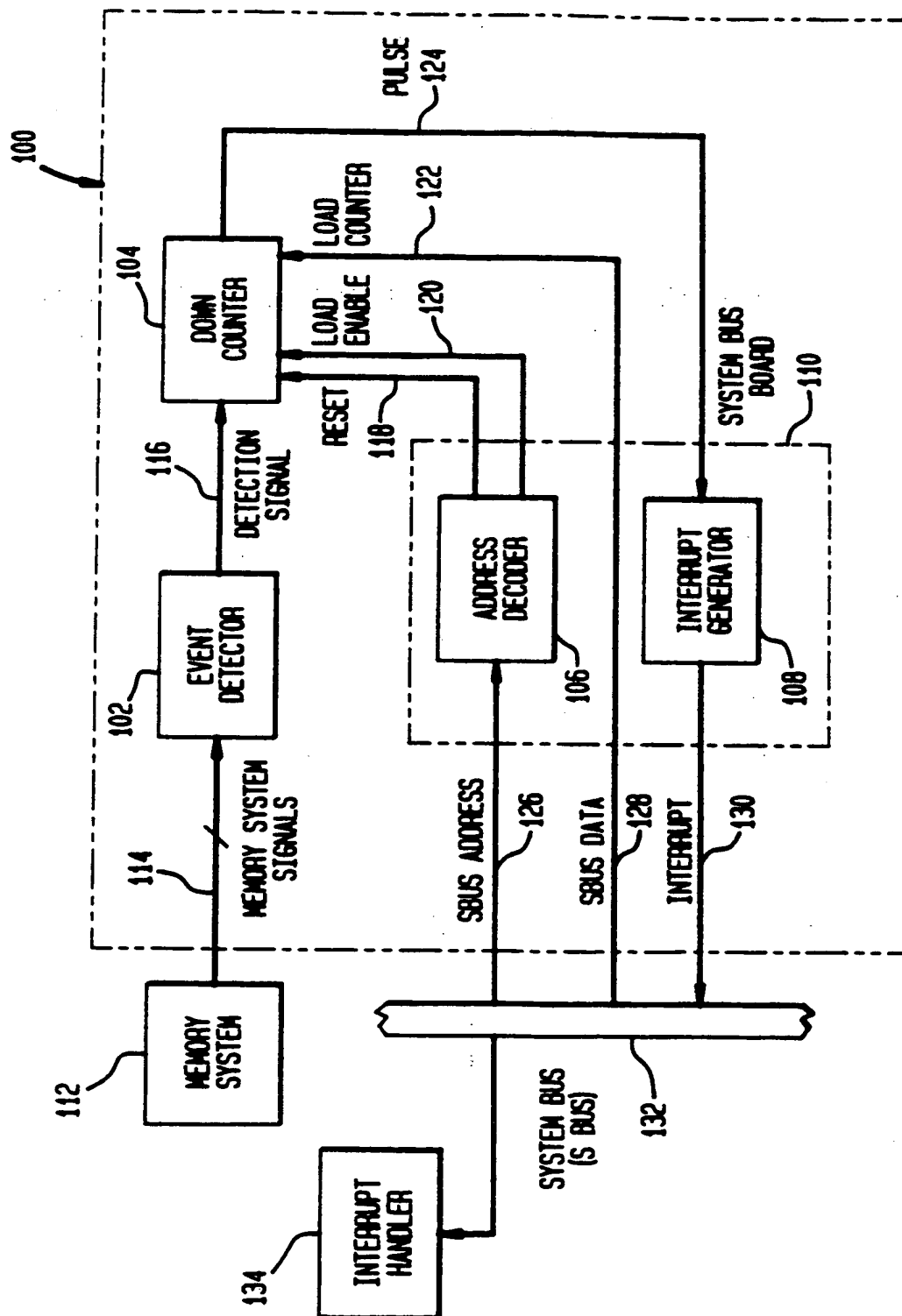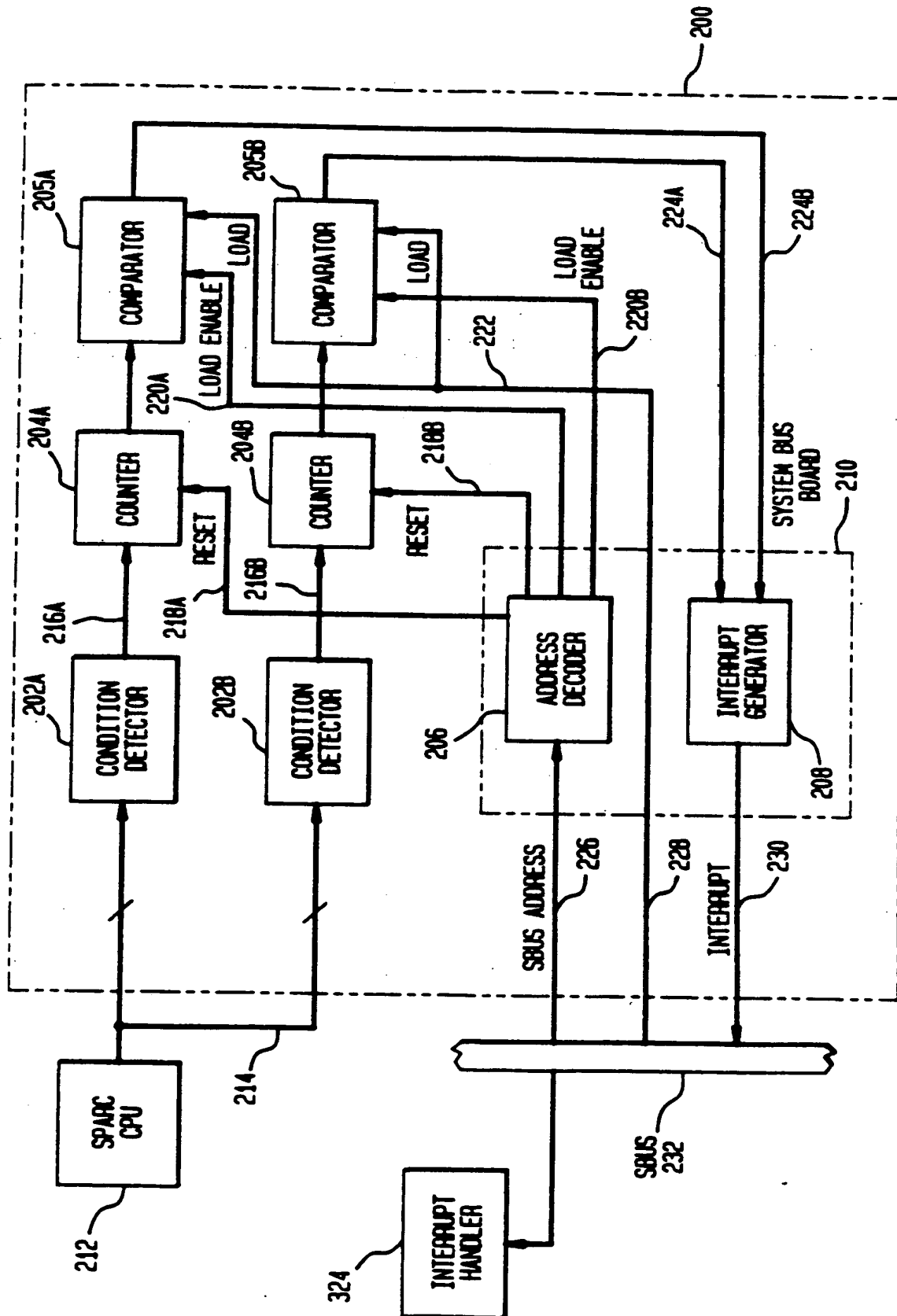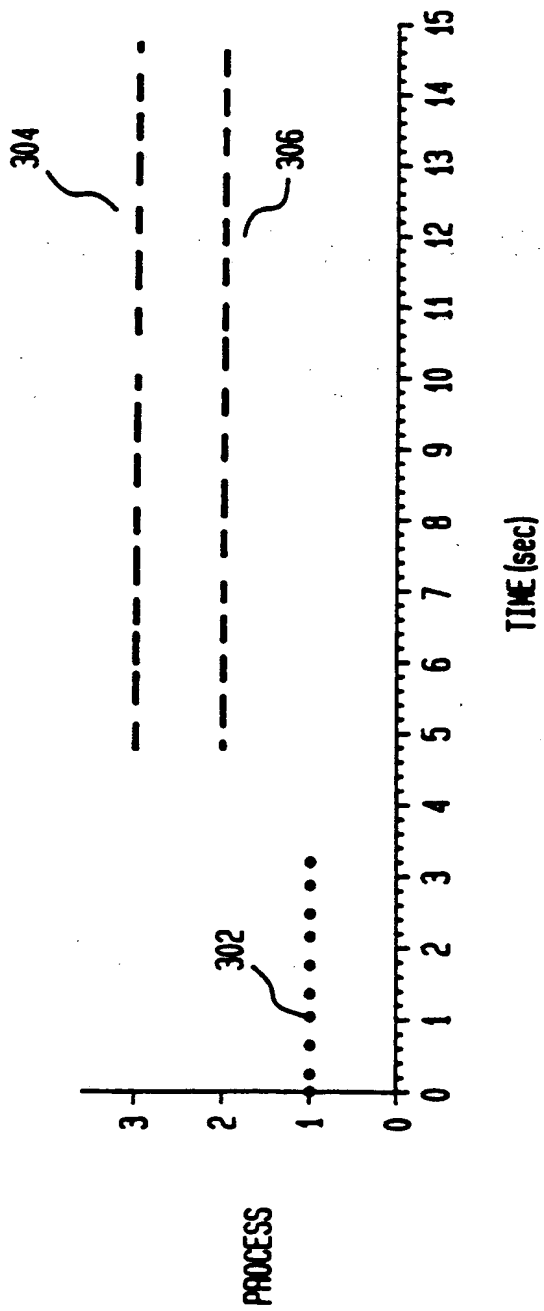
10

15

20

25

30

35

40

45

50

55

FIG. 1

FIG. 2

FIG. 3

(12) **EUROPEAN PATENT APPLICATION**

(72) Inventors:
• Agrawal, Prathima
New Providence, New Jersey 07974 (US)
• Trotter, John Andrew
Chatham, New Jersey 07928 (US)
• Goldberg, Aaron Jay
Princeton, New Jersey 08540 (US)

(74) Representative:
Watts, Christopher Malcolm Kelway, Dr.
Lucent Technologies (UK) Ltd,
5 Mornington Road
Woodford Green Essex, IG8 0TU (GB)

(54) **Interrupt-based hardware support for profiling system performance**

(57) In order to get empirical data on memory behaviour to guide system optimisation, a system and method for profiling computer systems is presented.

By augmenting a cache miss counter with a compare register and an interrupt line, the processor is interrupted when the counter matches the compare value, and the state of the processor is sampled. This can happen at a predetermined number of cache misses or cache stall cycles. Using these sampled data, a cache miss profile can be developed that associates cache misses with specific processes, procedures, procedure call stacks, or even user defined aspects of the system.

Also, the system is able to provide assistance in isolating performance bottlenecks and guiding optimisation of architectures, operating systems, compilers and applications.

FIG. 1

**European Patent Office**

# EUROPEAN SEARCH REPORT

**Application Number**

EP 95 30 3485

## DOCUMENTS CONSIDERED TO BE RELEVANT

| Category | Citation of document with indication, where appropriate, of relevant passages | Relevant to claim | CLASSIFICATION OF THE APPLICATION (Int.Cl.6) |
|---|---|---|---|
| P,Y | IBM TECHNICAL DISCLOSURE BULLETIN, vol. 37, no. 9, 1 September 1994, pages 589-592, XP000473517 "TECHNIQUE FOR SPEDULATIVELY SAMPLING PERFORMANCE PARAMETERS" <br> * page 589, column 25, line 29 * <br> * page 590, line 25 - page 591, line 3 * <br> --- | 1,3,4,6, 7,12,13, 15 | G06F11/34 |
| Y | IBM TECHNICAL DISCLOSURE BULLETIN, vol. 34, no. 7A, 1 December 1991, pages 210-217, XP000255534 "CO-PROCESSOR FOR HARDWARE-SUPPORTED MEASUREMENT AND MONITORING FUNCTIONS" <br> * page 212, line 7 - page 213, line 16 * <br> * page 215, line 1 - line 4 * <br> * page 216, line 19 - line 28 * <br> --- | 1,3,4,6, 7,15 | |
| Y | EP 0 141 357 A (HONEYWELL INC) 15 May 1985 <br> * figures 1,3 * <br> * page 12, line 1 - page 17, line 23 * <br> * page 5, line 24 - page 6, line 21 * <br> * page 4, line 19 - page 5, line 2 * <br> * abstract * | 7,12,13 | |
| A | --- | 1,3,4,15 | TECHNICAL FIELDS SEARCHED (Int.Cl.6) <br><br> G06F |
| D,Y | IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, vol. 4, no. 1, 1 January 1993, pages 41-61, XP000381324 LENOSKI D ET AL: "THE DASH PROTOTYPE: LOGIC OVERHEAD AND PERFORMANCE" | 6,13,15 | |
| A | * abstract * <br> * page 46, right-hand column, line 25 - page 47, left-hand column, line 3 * <br> * figures 4,5 * <br> --- | 1,12 | |

-/--

The present search report has been drawn up for all claims

| Place of search | Date of completion of the search | Examiner |
|---|---|---|
| BERLIN | 15 May 1997 | Masche, C |

EPO FORM 1503 03.82 (P04C01)

| European Patent Office | EUROPEAN SEARCH REPORT | Application Number EP 95 30 3485 |
|---|---|---|

## DOCUMENTS CONSIDERED TO BE RELEVANT

| Category | Citation of document with indication, where appropriate, of relevant passages | Relevant to claim | CLASSIFICATION OF THE APPLICATION (Int.Cl.6) |
|---|---|---|---|
| A | INTELLECTUAL LEVERAGE: DIGEST OF PAPERS OF THE SPRING COMPUTER SOCI INTERNATIONAL CONFERENCE (COMPCON), SAN FRANCISCO, FEB. 28 - MAR. 4, 1994, no. -, 28 February 1994, INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, pages 399-404, XP000479416 FRANKLIN M T ET AL: "POWER2: PERFORMANCE MEASUREMENT AND ANALYSIS OF TPC-C" * page 402, right-hand column, line 43 - page 403, left-hand column, line 26 * * page 399, right-hand column, line 37 - page 401, left-hand column, line 20 * * page 399, left-hand column, line 10 - line 16 * ----- | 1,12 | |
| | | | TECHNICAL FIELDS SEARCHED (Int.Cl.6) |

The present search report has been drawn up for all claims

| Place of search | Date of completion of the search | Examiner |
|---|---|---|
| BERLIN | 15 May 1997 | Masche, C |

EPO FORM 1503 03.82 (P04C01)